Programmation des Systèmes

Devoir surveillé intermédiaire

20 octobre 2015

Durée : 1h
Documents (notes de cours, mini-man, TD et TP) autorisés

Vous expliquerez votre code de sorte que votre intention (et pas seulement votre implémentation) soit claire. La gestion des erreurs pourra rester très simple, par exemple en interrompant le programme si nécessaire.

Les questions sont annotées par la longueur de la réponse attendue :

- courtes *(c)* : une ou deux phrases, 2 ou 3 lignes de C, etc.
- longues *(l)* : un ou plusieurs paragraphes, une fonction d'au moins 10 lignes, etc.
- moyennes *(m)* : entre les deux.

Le barème tiendra compte du temps nécessaire pour répondre.

1 Recherche d'un i-nœud

Voici le résultat de la commande `ls` affichant récursivement (`-R`) le contenu d'un répertoire en précisant pour chaque entrée son numéro d'i-nœud (`-i`) avant les autres informations (`-l`). `ls` a été déclenché dans le répertoire `/tmp/test` d'un périphérique de numéro 1234.

```
$ pwd
/tmp/test
$ ls -R -i -l
.:
total 16
4673 -rw-r--r-- 3 ray mond   10 oct 10 12:14 a
4673 -rw-r--r-- 3 ray mond   10 oct 10 12:14 b
4674 -rw-r--r-- 2 ray mond   10 oct 10 12:14 c
4675 lrwxrwxrwx 1 ray mond    1 oct 10 12:14 d -> a
4676 drwxr-xr-x 2 ray mond 4096 oct 10 12:14 sous-rep

./sous-rep:
total 8
4674 -rw-r--r-- 2 ray mond 10 oct 10 12:14 e
4677 lrwxrwxrwx 1 ray mond  1 oct 10 12:14 f -> e
4673 -rw-r--r-- 3 ray mond 10 oct 10 12:14 g
4678 lrwxrwxrwx 1 ray mond  4 oct 10 12:14 h -> ../c
4679 lrwxrwxrwx 1 ray mond  1 oct 10 12:14 i -> c
```

L'objectif de cet exercice est de proposer une fonction qui affiche l'ensemble des chemins qui correspondent à un i-nœud donné sur un périphérique.

Dans un premier temps, nous ne nous intéresserons pas aux liens symboliques. Ainsi, la recherche de l'i-nœud de numéro 4673 dans le répertoire `/tmp/test` (sur le périphérique 1234), effectuée en appelant `affiche_chemins(1234, 4673, "/tmp/test")`, devra seulement afficher :

```
a
b
sous-rep/g
```

Notez que `d`, qui est un lien symbolique qui pointe vers l'i-nœud cherché, n'apparaît pas dans cette liste.

Le système ne fournit aucun moyen d'accéder directement à un i-nœud ou à un chemin qui lui correspond à partir de son numéro. Nous allons donc devoir parcourir l'ensemble de l'arborescence pour retrouver ces chemins.

Q 1. *(c)* Expliquez pourquoi le système ne donne pas accès directement au contenu d'un i-nœud à partir de son numéro.

Q 2. *(c)* Expliquez pourquoi la fonction `affiche_chemins` prend en argument non seulement un numéro d'i-nœud mais aussi celui d'un périphérique.

Q 3. *(l)* Donnez en français, en les numérotant, les étapes à réaliser pour écrire `affiche_chemins`.

Q 4. *(c)* Définissez une fonction

```
void affiche_chemin_si_cherche(dev_t periph, ino_t inoeud, char *chemin)
```

qui affiche `chemin` si et seulement si ce chemin correspond à l'i-nœud cherché, c'est-à-dire celui dont le périphérique et le numéro sont donnés en argument.

Q 5. *(l)* Définissez la fonction récursive

```
void affiche_chemins(dev_t periph, ino_t inoeud, char *chemin)
```

Nous voulons maintenant aussi pouvoir suivre les liens symboliques, c'est-à-dire afficher les chemins des liens symboliques qui pointent vers l'i-nœud cherché. Ainsi, sur le premier exemple, `d` sera aussi affiché, avec `a`, `b` et `sous-rep/g`.

Q 6. *(c)* Sur l'exemple précédent, indiquez le résultat que l'on attend quand on cherche l'i-nœud de numéro 4674 en suivant les liens symboliques.

Q 7. *(c)* Indiquez ce qu'il faudrait changer à la fonction `affiche_chemins` pour qu'elle suive les liens symboliques. Expliquez au passage comment votre proposition d'implémentation traite les liens symboliques cassés.
On ne demande pas ici de réécrire le code complet !

2 Lectures « tamporisées »

Les appels systèmes sont coûteux en temps : ils prennent beaucoup plus de temps qu'un appel de fonction, par exemple. La bibliothèque standard fournit du coup une série de fonctions de lecture et écriture qui utilisent des tampons mémoires pour limiter le nombre d'appels systèmes effectués.

Prenons l'exemple de la fonction `fgetc` qui sert à lire un octet dans un fichier. Au lieu de lire un seul octet du fichier à chaque appel, elle lit d'un seul coup une séquence d'octets du fichier, séquence

qu'elle stocke dans un tampon mémoire. À chaque appel suivant de la fonction, elle retourne simplement un octet du tampon jusqu'à ce que tout son contenu ait été lu.

L'objectif de cet exercice est d'implémenter une fonction `kgetc` simple suivant ce principe général. Pour simplifier les choses, nous accéderons ici aux fichiers en *lecture seule*.

Votre implémentation utilisera bien entendu directement les appels systèmes (`open`, `read`, `close`, etc.), pas les fonctions de la bibliothèque standard utilisant des tampons mémoire.

Q 8. *(l)* Donnez en français, en les numérotant, les étapes à réaliser pour la fonction `kgetc`.

Q 9. *(m)* Définissez une structure de données `KFILE` en indiquant à quelle(s) étape(s) identifiée(s) à la question précédente est utile chaque champ.

Q 10. *(l)* Définissez les fonctions suivantes :

```
KFILE *kopen(char *chemin)
int    kgetc(KFILE *kfile)
void   kclose(KFILE *kfile)
```

où `kgetc` retournera `EOF` quand la fin du fichier est atteinte, et `ERREUR` si la lecture a échoué.

```
#define EOF    (-1)
#define ERREUR (-2)
```

Dans le cas général, nous voudrions bien entendu pouvoir effectuer des lectures et des écritures.

Q 11. *(l)* Expliquez, sans forcément donner du code, comment vous implémenteriez une interface permettant lecture et écriture, en précisant notamment quelles données sont nécessaires pour mener à bien les opérations.